AD 709224

ON THE IMPLEMENTATION
OF THE DESCRIPTIVE DATA BASE,
BASED ON CDL1

BY

CHITOOR V. SRINIVASAN

RCA LABORATORIES
PRINCETON, NEW JERSEY 08540

SCIENTIFIC REPORT NO. 4

FEBRUARY 1970

PREPARED FOR

AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD, MASSACHUSETTS 01730

DISCLAIMER NOTICE

ABSTRACT

In previous reports, CDL1 -- a computer description language -- has
been defined and discussed. This report discusses the implementation of a
system of programs, on the RCA Spectra 70 computers, to generate appropriate
file structures from computer descriptions written in CDL1. This trans-
lation to a DDB -- descriptive data base -- involves syntactic analysis
and a certain amount of checking for internal consistency, as well as the
creation of directory entries, etc. Once the type of DDB's described
in this report can be generated, a variety of design-aid systems can be
based upon them, saving a duplication of effort, guaranteeing an integrated
overall system, and avoiding built-in obsolescence.

The final state of the work done under this contract is given, with
details of the file and directory structures which have been implemented.

FOREWORD

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1. INTRODUCTION

I would like to outline in this report the implementation task ahead of us in the development of a <u>Descriptive Data Base</u> (DDB) for computing system descriptions, based on the language CDL1* [1,2]. Let me first explain what DDB is, what purpose it is to serve in a computer system design environment, and to begin with, how it may be used.

1.1 DDB, What It Is and How It Is to Be Used

The Computer Description Language, CDL1, provides basically the following two facilities:

(A) It provides a systematic way of describing situations that arise in a computer system design activity, at the level of <u>System Architecture</u> and <u>Logic Design</u>. This involves facilities to describe system and logical structures and tasks performed by them. The tasks themselves could be described either functionally (independent of their implementation), or in terms of <u>data-flow</u> <u>sequences</u> initiated by them within an abstract system (this would be specific to a given system architecture), or in terms of <u>control/command</u> sequences appearing within a system (specific to a given control structure). These different descriptions of a task would reflect in a natural way the different stages of design of a system, stages of partial definition of the logical structure of the system. As design progresses the total system description would get built up in terms of descriptions of the various sub-systems and their interfaces. In fact, the design process itself would be viewed as one of generating the successive levels of descriptions, in increasing levels of logical detail. In a design environment, this generative process would be a collective activity involving the participation of several designers.

(B) The language also provides a scheme for filing the descriptions so produced. Each body of description would contain implicit information on how it should be filed, and how it should be embedded within the body of description already in file. Thus, the way the descriptive file might grow would depend on the descriptions themselves.

The features of the language which provide for these facilities were discussed in earlier reports [1]. An overall introduction to the language appears in AFCRL-67-0565 [2] and the formal definition of CDL1 in AFCRL-67-0588 [3].

The essential task in the development of the DDB is one of producing the software for CDL1 File Establishment (CDLFE). The CDLFE process involves the following:

---

*CDL1 is a Computer Description Language.

i)   Syntactic analysis of input descriptions in CDL1 to validate them.

ii) Limited interpretation of the syntax for checking consistency of descriptions and their completeness (in some sense), for determining the directory entries to be made for a given body of description, and for encoding of a description into its internal format.

The data files so created, together with their directories would constitute the DDB (Descriptive Data Base). This DDB is, in fact, simply an elaborate symbol table. The symbols used in the DDB would denote very complex objects. Hence, to store and access the definitions pertaining to the symbols it would be necessary to have a data structure more complex than that of a simple table. The details of this data structure are inferred from the descriptions themselves. Figure 1 shows the block diagram of the CDLFE process. The figure is self-explanatory.

The DDB would establish a common context within which a variety of automatic design aids could be developed. Each such design aid system would communicate with the DDB via a Data Retrieval and Abstracting Facility. The development of the description language has led us not only to identify the kinds of system design aids that one could develop, but also has pointed the way to some novel techniques of realizing them.

1.2 How I Propose to Use DDB Now

The DDB together with the Data Retrieval (DR) facility would constitute the Design Documentation System (DDS). Figure 2 shows a block diagram of the DR facility. This figure is self-explanatory too. The DDS as presently conceived would not only establish a foundation for developing a variety of design aid systems of the future but, more importantly, would be of immediate use and benefit to all current system design efforts, purely as a powerful (flexible), mechanized, central documentation facility for all designs. The value of having such a central documentation facility cannot be overemphasized.

Once the DDB is created, work on the development of DR could proceed simultaneously with several other design automation efforts. For example, one could easily undertake the development of data-flow simulators (EO-simulators) in the context of the DDB, as also functional simulators at a level higher than the data flow. The availability of DDB would make it possible to think in concrete terms about the various design aid systems. Also, it would provide the guarantee that the systems so developed could all be integrated into a central design aid facility.
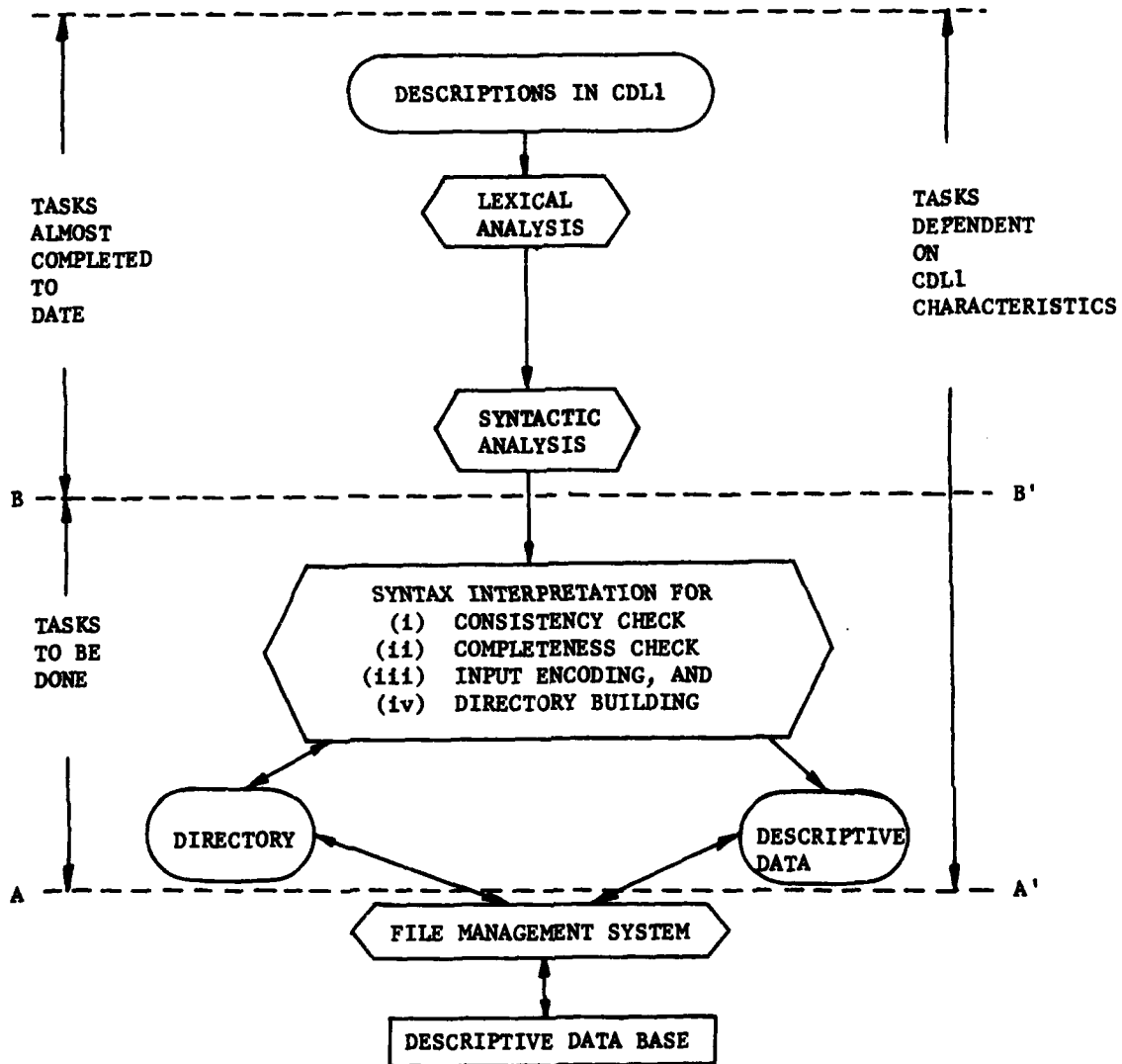
2

Figure 1. Block Diagram of CDL1 File Establishment Process

```
         ┌─────────────────────────────┐
         │  DESCRIPTIVE DATA BASE      │
         └─────────────────────────────┘
                        ↑↓
                ┌──────────────┐
                │ FILE         │
                │ MANAGEMENT   ├────────→  ┌──────────┐
                │ SYSTEM       │           │  OUTPUT  │
                └──────────────┘           └──────────┘
                        ↑↓

DATA RETRIEVAL SYSTEM
┌──────────────────────────────────────────────────────────────┐
│   ┌─────────┐     ┌──────────┐      ┌──────────┐              │
│   │ QUERY   │     │RETRIEVAL │      │DIRECTORY │              │
│   │PROCESSOR├───→ │COMMANDS  ├───→  │SEARCH    │              │
│   └─────────┘     └──────────┘      │FACILITY  │              │
│                                     └──────────┘              │
└──────────────────────────────────────────────────────────────┘
                        ↑
                ┌──────────────────┐
                │ QUERIES IN A     │
                │ RETRIEVAL LANGUAGE│
                └──────────────────┘
```
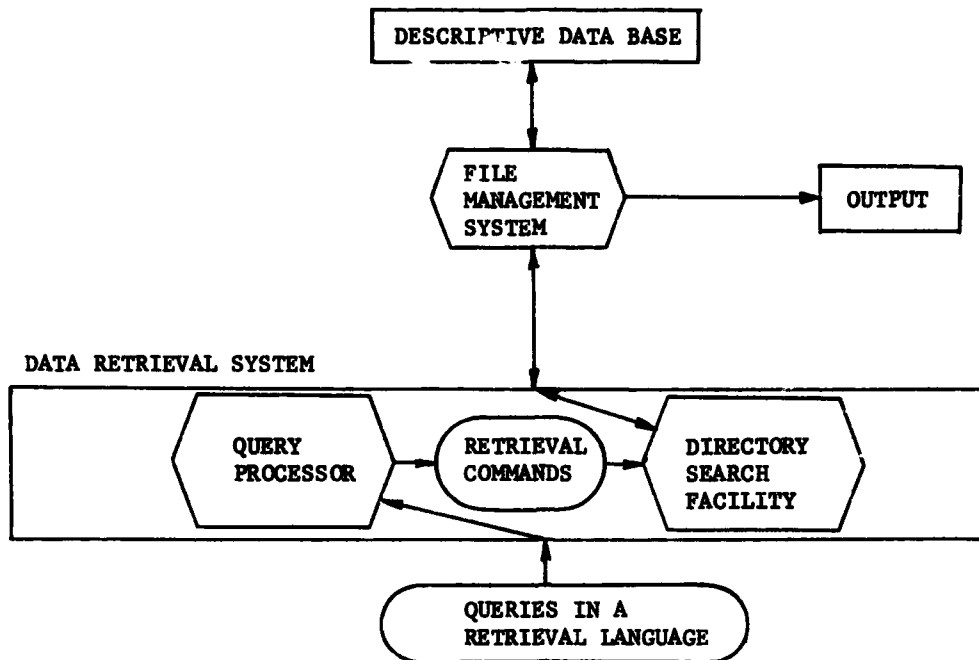
Figure 2. Block Diagram of the Data Retrieval Facility

The development effort necessary to create such a DDB would be far less than
what would be necessary if one chose, for example, to develop a data-flow
simulator starting from scratch. I think one should be very careful about
choosing design automation projects conceived in isolation outside the
context of a general design aid system, for such projects have the disadvantage
of built-in obsolescence. Beginning the design automation effort with the
construction of DDB appears to me as a very sensible approach from the view-
point of technical feasibility, practical needs, and future promise.

Let me now explain some of the details involved in the CDLFE procedure.

4

2. THE CDL1 FILE ESTABLISHMENT (CDLFE) PROCESS

2.1 The Present State of Our Work

The tasks to be performed in the CDLFE process are shown in Figure 1.
Those appearing above the dashed line AA' in Figure 1 are directly associated
with CDL1 and depend totally on CDL1 characteristics. The file management
task, appearing below the line AA', could be part of the operating system in
which CDLFE would function. I shall, therefore, confine my discussion here
only to the tasks appearing above AA'.

The software for the lexical and syntactic analysis of CDL1 strings is
now almost ready. The syntactic analysis is specified only for a subset of
CDL1. (It excludes all expressions in the language and only the general forms
of CDL1 statements and declarations are included.) The detail of the analysis
would be just sufficient for the purposes (ii), (iii) and (iv) indicated in
Figure 1 within the 'SYNTAX INTERPRETATION' block. The lexical and syntactic
analyzers have been individually debugged, and have had a single successful
run together. Further debugging should be done. Also, the syntax table needs
a few additions and modifications. The block diagram of the routines is shown
in Figure 3. The syntax table for the CDL1 subset was produced by the LR1
processor written by A. J. Korenjak [4,5]. This technique of syntax table
generation has incidentally established CDL1 to be a deterministic language
(parsable without backup). All software packages shown in Figure 3 were
written in the SPECTRA assembly language. The box with dashed lines in the
figure is yet to be described.

The lexical analyzer, together with the hash and associated tables and
hash routines, occupies about 11 000 bytes. This area includes the buffer
blocks for future directory building. The syntax tables occupy about 28 000
bytes, and the stack manipulator occupies about 580 bytes.

The major unfinished task ahead of us is syntax interpretation. It
should be noticed that the interpretation is not for producing an output code
in an object language as is done in a compiler, but it is for finding out how
a given body of description should be filed. The consistency and completeness
checks sould be incidental to file establishment. For the present we have
planned to implement only the completeness checks for the definitions in the
language. To check consistency it would be necessary to have a greater degree
of syntax analysis than what has been presently implemented. The information
for building the directory is obtained almost entirely from the declarations
in the language. The part of the directory not determined by the declarations
is dependent upon the module structure* of the descriptions, and the statements
appearing within them. The design specification for the DDB is now complete.
This involves mainly the design of the directory for accessing the DDB.

---

*Modules are basic units of descriptions in the language.
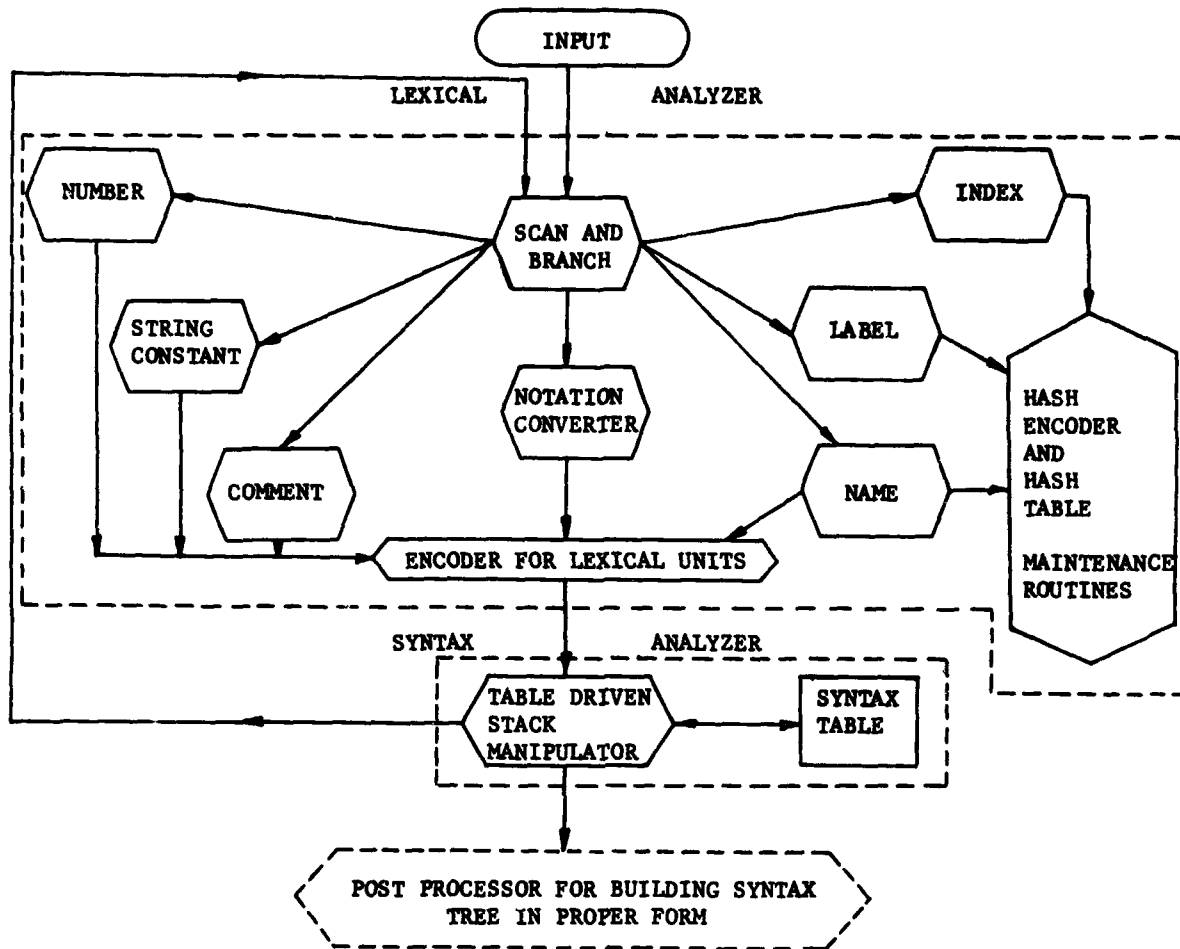
Figure 3. **Block Diagram of Lexical and Syntax Analyzers for the CDLFE Process**

6

The organization of the software packages necessary to produce the DDB, starting from the parse tree of descriptions in CDL1, has not yet been decided upon. The entire DDB establishment software is going to be written in terms of macros. A suitable collection of macros for the purpose is now being defined in SNOBOL. These macros, in fact, would constitute a special-purpose implementation language for the CDLFE processor. The software written in the macros would be, in a sense, machine-independent. To transfer the system software to another machine it would be only necessary to redefine the macros. We believe that as an implementation tool these macros would be very useful in a variety of system software implementation tasks.

Let me now briefly describe the DDB directory structure.

2.2 DDB Directory Structure

(A) File Structure

The DDB file is a page-organized file. Each page is 2048 bytes long and has seven attributes associated with it. These attributes are shown in Table I. The attribute value of a page is indicated by the page flag which is eight bits long. The bits of the page flag are set to 0 or 1 depending upon the values of its attributes, in the order shown in Table I. This flag is part of the page header, shown in Figure 4.

TABLE I
PAGE ATTRIBUTES

Bit No. in the Page Flag	Attribute Name	Attribute Values	
		0	1
0	TYPE	data	directory
1	Version	old	new
2	Changes	no	yes
3	State	clean	dirty
4	Record Length	variable	fixed
5	Size	single	double
6	Vacancy	yes	no

The various fields in the page header are intended for the following purposes:

(i) VP#. This is the Virtual Page Number of the page. VP# identifies a page uniquely. This header entry may be used to verify a page identity when it is brought into core.

(ii) Page Flag. This denotes the page classification according to its attributes, shown in Table I. The attributes have the following significance:

a) Type: Directory pages are handled differently from the data pages for storage and updating.

7
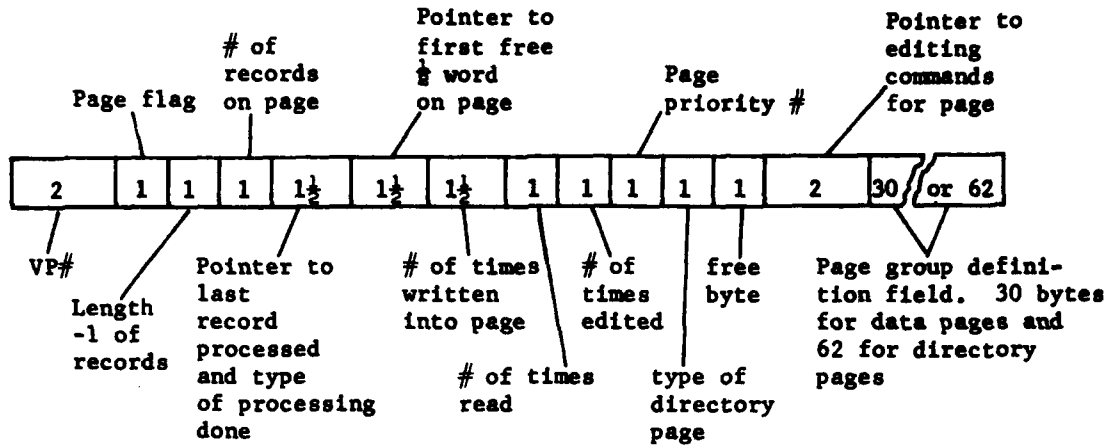
Figure 4. Page Header Format. The numbers within the boxes indicate the size of the box in bytes (8 bits/byte)

b) **Version:** If the version of a page is OLD then it needs updating.

The editing commands for updating the page may be stored separately in a different page. The pointer to the editing commands would appear in the last 2-byte field of the header, just before the group definition field (see Figure 4). If the editing commands are used on the page then the version of the page would change to NEW.

c) **Changes:** This indicates whether a page in core had been written into or not. If the value of this attribute is YES, then the page should be stored back on the disc before it is dropped from the core; otherwise, it may be dropped without disc transfer.

d) **State:** A dirty page requires garbage collection and a clean page does not.

e) **Record Length:** The significance is obvious.

f) **Size:** In certain cases, two pages may be treated together as though they were one large page. This attribute indicates the appropriate size of the page.

g) **Vacancy:** The significance is obvious.

(iii) **Length Field:** For pages with fixed-length records the record length is stored in this field. The maximum permissible fixed-length record length is 256 bytes.

8

(iv) <u>Record Count</u>: The number of records on a page is stored here.

(v) <u>Last Record Pointer</u>: Records within a page should always start at a 1/2-word boundary. The last ten bits of this pointer point to the record last processed in the page. The first two bits of the pointer specify the type of processing done, which could be one of the following three: Read, Write, or Edit.

(vi) <u>Free Word Pointer</u>: This points to the first free 1/2 word in the page, beginning at a 1/2-word boundary. The available space on a page is documented within the page as a list. The first free 1/2 word pointed to by this pointer would specify the count of available free bytes beginning at the 1/2-word boundary, and in addition would contain a pointer to the next free 1/2 word in the page starting at another 1/2-word boundary. This process is iterated until all the available space in the page is exhausted.

(vii), (viii), (ix) and (x) <u>Counts of Page Usage</u>: The various counts of the way the page was used would determine its priority number. The priority number so produced may be used for page allocation in a memory hierarchy.

(xi) <u>Directory Type</u>: In the case of a directory page the type of directory is identified by this field. The directory format depends on its type.
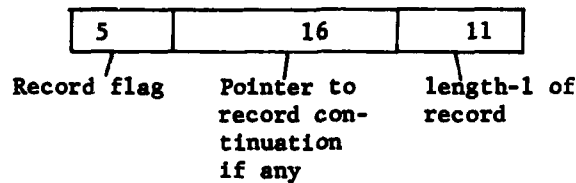
(xii) <u>Edit Pointer</u>: This was referred to earlier in item (ii)b. If a user wants to edit a page without destroying its existing contents the page may be tagged as OLD and the editing commands for the page may be stored in one of its slave pages. The edit pointer field would then have a pointer to the edit commands so specified. Every time the page is called one may then call for either its OLD or its NEW version.

(xiii) <u>Page Group Definition</u>: Each page may have a group of slave pages associated with it. A data page may have up to 15 slave pages, and a directory page, up to 31 slave pages. The data within a master page may contain pointers to the records in any one of its slave pages. The pointer format is:

5	11	BITS

Slave page #   Displacement in
                slave page
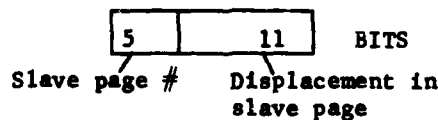
For $1 \leq i \leq 31$ (or 15) the Virtual Page # of the ith slave page would be at the ith half-word boundary of the Page Group Definition field of the header. This group structure of pages enables one to economize on memory at the expense of a slight increase in computation, in a situation where cross-referencing among pages is confined to page groups. It should be noted that every page in the file would be the master page of its associated group of up to 15 (or 31) slave pages. Thus, the master page of one group would itself be a slave page of another group and vice-versa.

9

Each variable length record in a page would have a 4-byte record header, with the following format:

5	16	11

Record flag   Pointer to   length-1 of
                    record con-   record
                    tinuation
                    if any

A record may be either complete or incomplete, and it may or may not have a continuation elsewhere within its page group. The first two bits of the record flag indicate these attributes. A record is considered to be complete if it is a well-defined statement in CDL1. Records may be left incomplete at one time and be completed at a later date. Complete records within a page need not appear in one continuous set of successive bytes. Whether a record is complete or not, it may appear in several segments within a page group. These record segments would be linked together as a list via the record continuation pointer in the record header. Thus, each segment of such a record would have its own header, and would itself have the status of a record within the page. The last segment in such a list of segments would point to the first segment of the record. The length field of the record header would always contain only the length of the segment associated with the header.

When a page is full, it is said to be dirty if one or more of the records in the page remain segmented. In such a case the page is due for garbage collection (or clean up). A clean page would not thus contain segmented records.

(B) Directory Structure

The directory for searching the DDB is organized around a central hashing scheme. This hashing scheme and its associated tables are used to translate a given key (which could be a name, label, or some values of qualitative attributes associated with the objects in DDB) to its corresponding address. This address would be a pointer to the relevant items within the DDB denoted by the key. The items denoted by a key either could be some descriptive passages within the DDB or could themselves be some entries in another directory of DDB leading to further search within the DDB. All searches within the DDB would necessarily begin with the hashing scheme. Let me, therefore, first describe this scheme.

B-1. The Hashing Scheme

A schematic diagram of the address translation process through hashing is shown in Figure 5, which may be interpreted as follows:

Explanation of Figure 5

The hash search scheme makes use of three components:

    i)   Hash Reference Table (HRT)
    ii)   Variable Length Items Bin (VLIB)
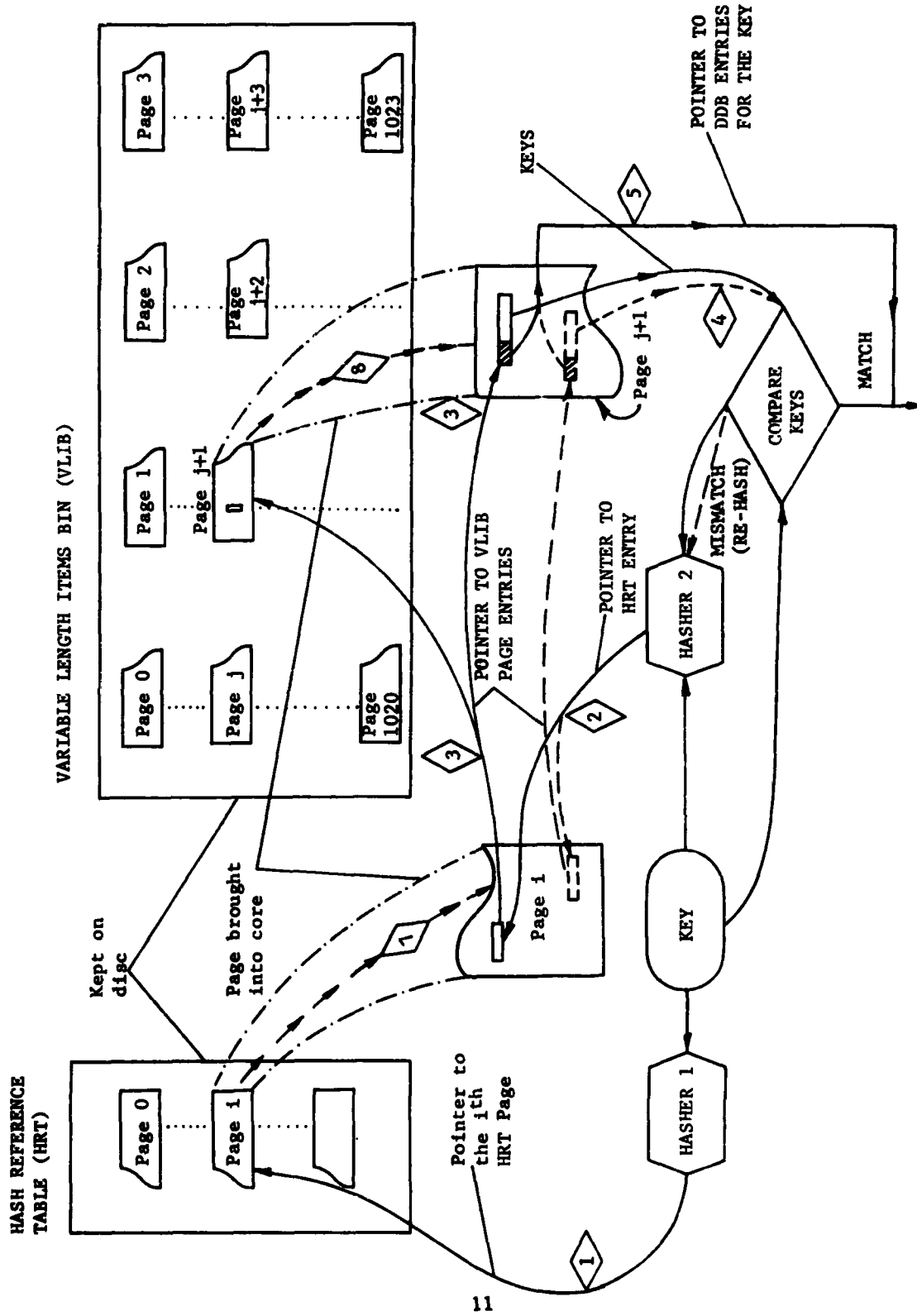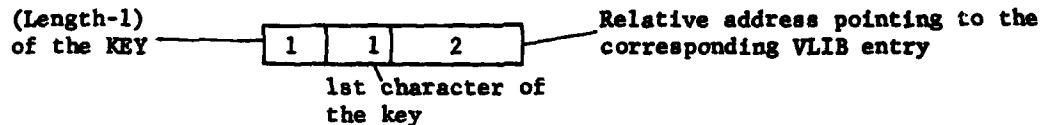   iii)   The Hashing and Comparison Routines

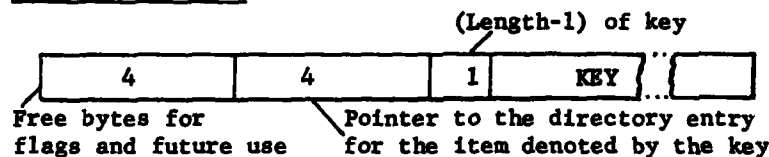10

Figure 5. A Schematic Diagram of the Hash Search Scheme

The search scheme is the following: a given key (the input key) is hashed both by HASHER 1 and HASHER 2 shown in Figure 5. The first hasher comes up with a 1 byte number ≤ 225, and the second one, with a number ≤ 503, which is a prime number. The output of the first hasher points to one of the 256 HRT pages (as indicated by arrow ⊕ in Figure 5), which is then brought into core (as shown by arrows ⊕ in Figure 5). Each entry in a HRT page is a 4-byte item, with the following format:

HRT Entry Format: (4 bytes)

(Length-1) of the KEY ———— [1 | 1 | 2] ———— Relative address pointing to the corresponding VLIB entry

1st character of the key

Each VLIB entry would have the format shown below:

VLIB Entry Format:

(Length-1) of key

[4 | 4 | 1 | KEY ... |]

Free bytes for flags and future use   Pointer to the directory entry for the item denoted by the key

Assuming an average length of 24 bytes per VLIB entry, one would, on the average, need about $(503 \times 24)/2048 \cong 6$ VLIB pages for each HRT page. The 503 entries in a HRT page would occupy $503 \times 4 = 2012$ bytes, leaving a remainder of 36 free bytes per HRT page. These 36 free bytes are used to store the VP#'s of up to 15 slave pages associated with the HRT page, and part of other page header entries. The slave pages of a HRT page would be either the VLIB pages or continuation pages for the HRT page. These continuation pages are used to take care of HRT page overflow.*

The output of the second hasher would point to a full-word boundary within the HRT page just brought into core, as shown by the arrow ⊕ in Figure 5. The HRT entry at this full-word boundary is the item of interest to us. Let us call it HRTE. The length and first character of key in the HRTE are compared with the length and first character of the input key. If they do not match, then the input key is rehashed by the HASHER 2 and the new hash address so obtained is again used similarly. This kind of probing of the HRT page is iterated until a successful match is obtained. This secondary probing is indicated in Figure 5 by the broken arrow coming out of arrow ⊕ . Let us call this process the initial matching process. If after 251 such probings no successful initial match is obtained, then the input key is interpreted as a new key not yet entered in the hash tables.

When a successful initial match does occur, the two-byte pointer in the HRTE with the successful match is used to bring out of VLIB the appropriate VLIB page, as indicated by the arrows ⊕ and ⊕ in Figure 5.

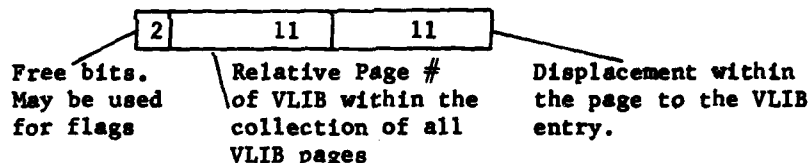*The overflow handling technique is not discussed in this report.

12

The full key in the appropriate VLIB entry, in the VLIB page just brought into core, is now compared with the input key. Notice that by this time we are sure that the length of the input key and its first character agree with the length and first character of the key in the VLIB entry. Therefore, the chances of now obtaining a full match are quite good. Let us call this the secondary matching process. If the secondary match is not successful then the input key is also rehashed by HASHER 2 and the entire process is repeated. The secondary matching process is iterated until a successful initial and secondary match is obtained. Here also, if after 251 such probings no match occurs then the input key is interpreted as being new.

Because of the way the HRT and VLIB tables are built up one can be reasonably sure that all secondary match attempts would be confined to the same VLIB page initially brought into core.

Once a successful secondary match is obtained, the 4-byte pointer in the VLIB entry with the successful match is chosen as the DDB address for further search and retrieval of the objects denoted by the key.

The HRT as presently configured can accommodate address translation for up to 503 x 256 = 128768 keys. For each one of these keys the address of its VLIB entry is called the Hash Reference Code (HRC) for the key. This HRC is a 3-byte item with the following format:

HRC FORMAT:

| Free bits. May be used for flags | Relative Page # of VLIB within the collection of all VLIB pages | Displacement within the page to the VLIB entry. |

This HRC will be used throughout the DDB for the identification of the key. Thus, the HRC of a key may never be changed.

A given key in CDL1 may denote more than one item in the DDB. To identify an item uniquely in the DDB it is generally necessary to have at most two keys: one of these two keys would be the name, label, or attribute of an object, and the other key would be the name or label of the scope of the object. Thus, a key in CDL1 would be unique only within a scope. The DDB would have thousands of such scopes. Hence, the number of objects denoted by these keys could be many times greater than the total number of keys.

CDL1 has two kinds of scopes: The first kind of scope is the scope of modules*, which are identified by module titles (names). The second

_____

*See Ref. 1 for a discussion of modules and module types.

13

kind of scope is that of _descriptive and interpretive_ blocks. These are similar to the BEGIN-END blocks in ALGOL, and are identified by labels*, or names of items interpreted. Both these scopes may have a hierarchical structure of _subscopes_. Thus, the name or label used to identify a scope would be its _tree name_ within this hierarchy. In the case of scopes the VLIB entry for the key identifying the scope would have a pointer to its corresponding scope directory. The scope directory would be used to locate all the objects in DDB associated with the scopes. In the case of module scopes, the _module type_* would determine the submodules it might have and the directory for the submodules.

In the case of labels the scope of a label is restricted to its block or module. In the case of names of declared items the scopes of such items would permeate all submodules or sub-blocks of the block or module in which the item is declared. Also, in certain cases the scopes of names would be _global_, i.e., throughout the entire descriptive file the name would uniquely denote the same item. The _scope directory_, _declarations directory_, and _label directory_ are organized to indicate these various scope inclusion properties.
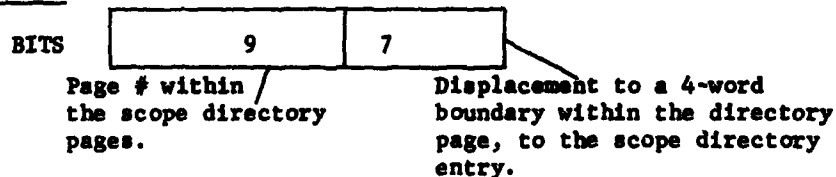
The descriptive items associated with a name depend upon the definition _format_* of the kind of object denoted by the name. It should be possible to locate for each name all its associated definitions, attributes, values, functions, interpretations, alternate definitions, alternate names, restrictions, etc. The _declarations directory_ would contain the relevant pointers to accomplish such identifications.

Let me now briefly describe how these three directories are organized. Together with the hash search scheme these constitute the core of the entire DDB structure. In addition to these directories, there are several others in the DDB, such as _user directory_, _name usage directory_, _item relationship directory_, _retrieval functions directory_, _definition format directory_, and _declaration format directory_, etc. I shall not describe these in detail in this report.

B-2. The Scope Directory (SD)

Each scope in DDB has a unique name, and is identified within the DDB by a unique 2-byte _scope number_ having the following format:

Scope Number Format:

BITS	9	7	

Page # within the scope directory pages.

Displacement to a 4-word boundary within the directory page, to the scope directory entry.

---

*Please see Ref. 1.

This scope number is used in all cross references within the DDB. Each scope directory entry is a 48-byte item with the format shown in Figure 6. The various fields in this directory entry have the following significance:

i) HRC: This is the Hash Reference Code for the scope name or label.

ii) Alternate Identifier: Each scope in DDB could have alternates described for them. These alternates might contain either several versions of design for a given item or alternate descriptions of a like design. Each scope may have up to 15 alternates, which are identified by numbers 1 through 15, and up to 16 which are identified by alternate names. Such alternate scopes would be denoted by the naming scheme:

"Label or Name of Scope ///ALT(# or name)".

iii) Scope Type: If the scope is a module then the scope type would be the same as the module type. Otherwise, the scope type would be one of the following:

Operation Interpretation Block,
Macro Definition Block
Function Interpretation Block,
Formal Definition Block,
Declaration Block, or
Begin-End Block.

iv) Scope Flag: The flag would indicate the information shown in Table II by virtue of its bit values. Several bits in the flag have been left undefined for future use.

The remaining fields point to the various directory entries associated with the scope. The pointers to parent/sibling and descendent scopes, in effect, specify the hierarchical structure of the scopes.

TABLE II
SCOPE FLAG

Bit #	Bit Values	
	0	1
0	No Alternates	Has Alternates
1	No Restrictions	Has Restrictions
2	No User Restrictions	Has User Restrictions
3	Has No Errors	Errors Exist
4	Needs Editing	No Editing
5	No Functions on Scope	Functions on Scope
6-15	NOT	USED

A module scope could have two kinds of descendent scopes, a descendent sub-module or a descendent block. A block scope cannot have as its descendent
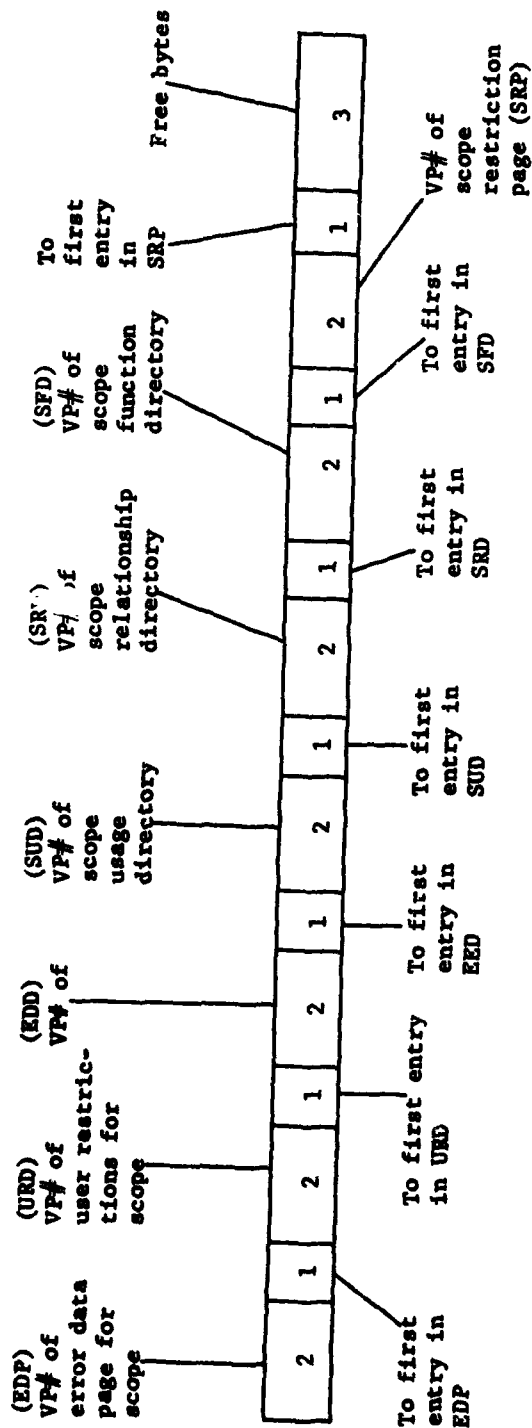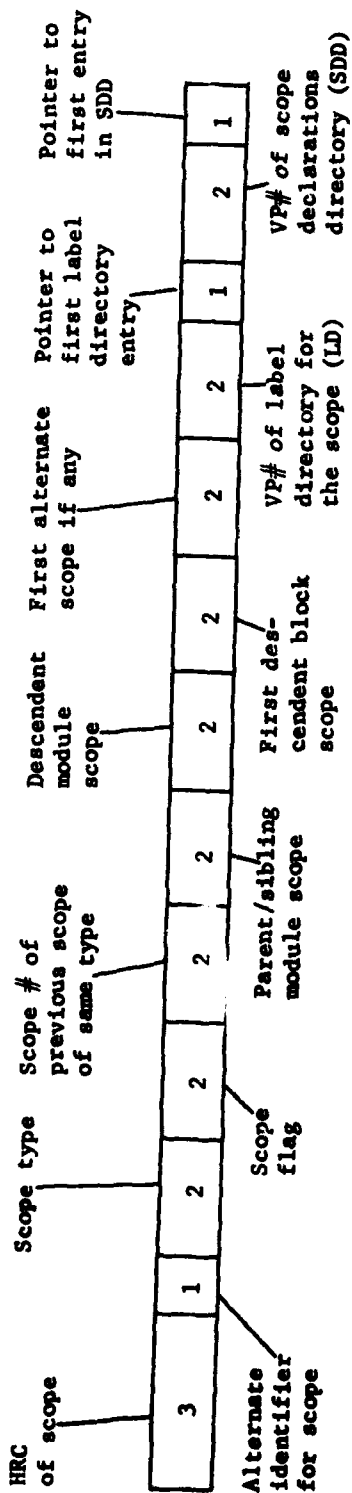
Figure 6. Scope Directory Entry Format

First entry format:

HRC of scope — Scope type — Scope # of previous scope of same type — Descendent module scope — First alternate scope if any — Pointer to first label directory entry — Pointer to first entry in SDD

Alternate identifier for scope — Scope flag — Parent/sibling module scope — First descendent block scope — VP# of label directory for the scope (LD) — VP# of scope declarations directory (SDD)

Box sizes: 3, 1, 2, 2, 2, 2, 2, 2, 1, 2, 1

Second entry format:

(EDP) VP# of error data page for scope — (URD) VP# of user restrictions for scope — (KDD) VP# of — (SUD) VP# of scope usage directory — (SR) VP# of scope relationship directory — (SFD) VP# of scope function directory — VP# of scope restriction page (SRP)

To first entry in EDP — To first entry in URD — To first entry in EED — To first entry in SUD — To first entry in SRD — To first entry in SFD — To first entry in SRP — Free bytes

Box sizes: 2, 1, 1, 1, 2, 1, 2, 1, 2, 2, 1, 3

16

a module; it could, however, have other blocks as its descendents. The descendent blocks of a module are separately identified, with special pointers, in the scope directory.

B-3. Label Directory (LD)

Labels are used in DDB to identify records (CDL1 statements), and blocks within a file. A record or a block in DDB is uniquely identified by its label and the name of the scope in which it occurs. Labels in DDB could themselves have a tree structure, as is evidenced by its directory entry, shown in Figure 8. To get at a record or a block in DDB it is necessary to use two Keys: its scope name and its label. The schematic search path for this is shown in Figure 7.

The VLIB (Variable Length Items Bin) entry for the scope name would contain a pointer to the scope directory entry for the scope. This scope directory entry itself would contain a pointer to the Label Directory (LD) page associated with the scope. The VLIB entry for the label would point to the head of a list within this LD page. This list in turn would contain the label directory entry for the label in search. By searching this list the directory entry for the label might be identified.

Notice that a same label might appear in DDB in several different scopes, each one of which would have its own associated label directory page.
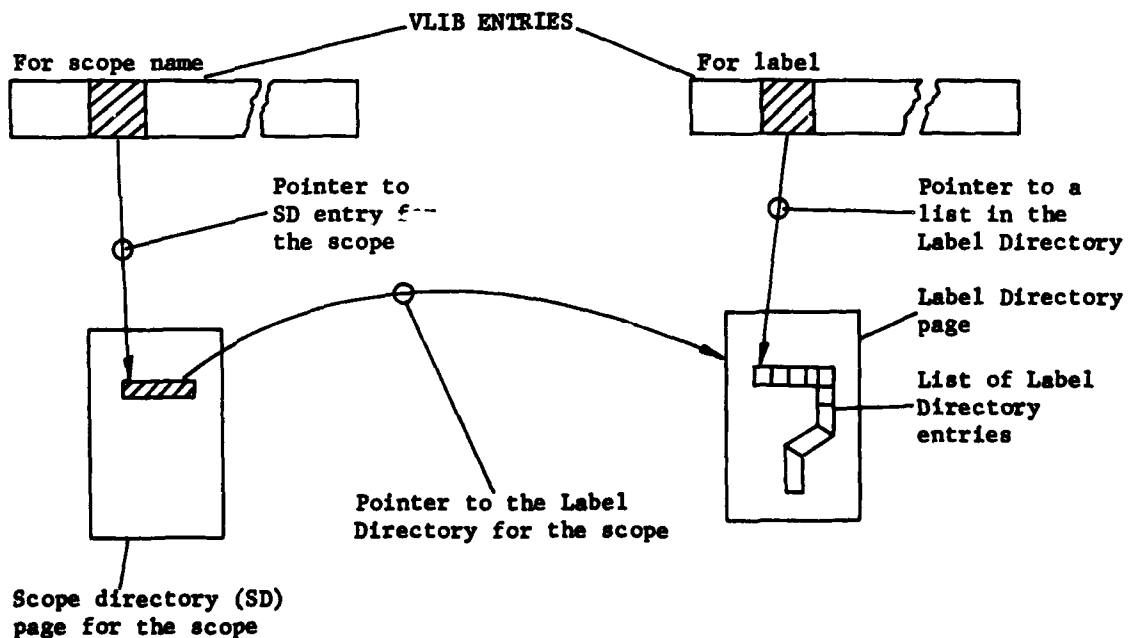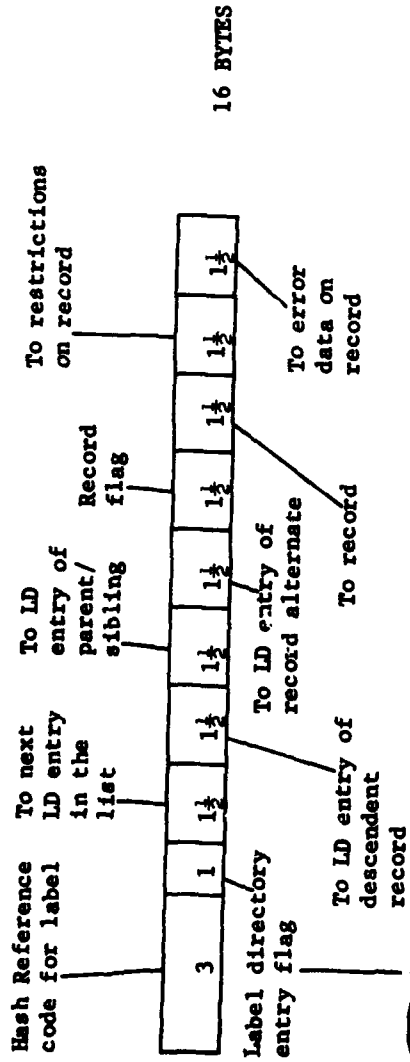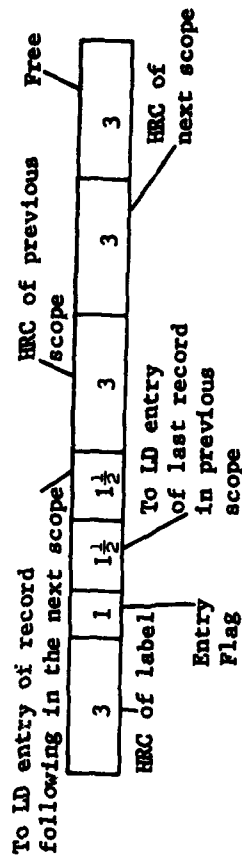

Figure 7. Schematic Diagram of Record or Block Search in DDB
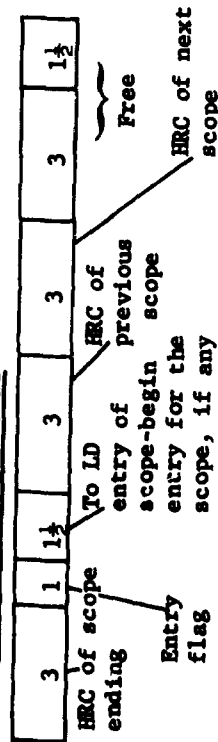
17

a) Record Entry Format

b) Scope Begin Entry Format

c) Scope End Entry Format

d) Alternate Record Entry Format

Same as a) except that the HRC code for label is replaced by pointer to original record.

Figure 8. Entry Formats for the Label Directory

18

In all these LD pages a search for a label would begin at the head of a list pointed to by the VLIB entry for the label. The relative address of this head of the list within the LD page would be the same, for a given label, in all these various LD pages. This head address is obtained by hashing the label.

A label directory entry for a new label in a scope is made as follows: First, the LD page for the scope is identified from the scope name; then from the existing VLIB entry of the label (if none exists then a VLIB entry is created), the pointer to the head of the list associated with the label is obtained. If the location of this head is empty in the particular LD page, then the LD entry for the label is made right at the head of the list. Otherwise, the LD entry is suffixed to the tail of the list already present.

The label directory entry formats are shown in Figure 8. The entries are all 16 bytes long. There are four different kinds:

a) Record Pointer
b) Scope Begin Entry
c) Scope End Entry
d) Alternate Record Entry

The various fields in the formats have the significance shown in Figure 8. All cross-referencing addresses within the LD have the following format:
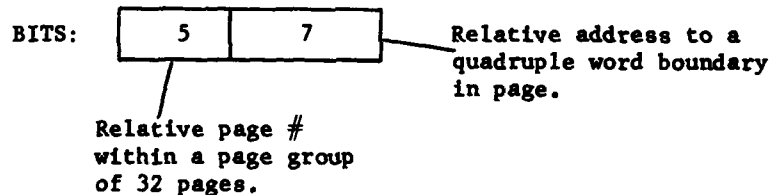
Each LD page may contain labels for up to four different module scopes. These modules are assigned local numbers 0,1,2, and 3 within an LD page; the local numbers are part of the entry flag for each LD entry, as shown in part a) of Figure 8.

B-4. Declarations Directory (DD)

This is used to refer to objects declared within descriptions in different scopes. An object in the description is identified uniquely by its name and the scope of the name. The directory entry for a name in the Declarations Directory (DD) is accessed very much in the same way as a Label Directory (LD) entry. From a given name of an object it is necessary, in general, to access the following associated information:

1. **Name Scopes:** Scopes in which the same name has been declared.

2. **Name Usage:** Scopes in which each token declaration of the name has been used.

3. **Definitions:** The definitions associated with the name. These would depend on the kind of name being defined.

4. **Initial Value:** Sometimes names might be assigned initial values.

5. **Current Value:** The current value of the name.

6. **Attribute Values:** Values of attributes associated with the name.

7. **Aliases:** Names which are aliases to the name.

8. **Author** of the name.

9. **Functions** on name.

10. **Conditions** on the declaration of a name.

11. **Restrictions** on a name.

Each item in a Declarations Directory (DD) page is a list, just like the lists in an LD page. The pointer in the VLIB entry of a name would point to the head of the list in DD. The search for a DD entry for a given name would be exactly as the search for a LD entry of a label, discussed in the previous section. An object in DDB would be uniquely identified by its name and the scope of the name.

The VLIB entry for a name would contain an additional pointer, which would point to a list containing all the scopes in which the name had been declared.

The DD entries fall into two classes: One for names and another for aliases (alternate names) given to an initially declared name. The formats for these two entries are shown in Figures 9(a) and (b). All cross-referencing within DD pages would have the same address format as those used in LD pages. Also, as in the case of LD, each DD page may contain declarations for up to four different scopes. These scopes are identified locally within a DD page by their local scope numbers.

The Scope Directory (SD), Label Directory (LD) and the Declarations Directory (DD) are the three principal directories used in DDB. Besides these there are several other directories to which entries in SD, LD, and DD point. I have not discussed the details of these other directories in this report. The hash search scheme, and the structure of SD, LD, and DD, provided the basic foundations on which DDB would be built.
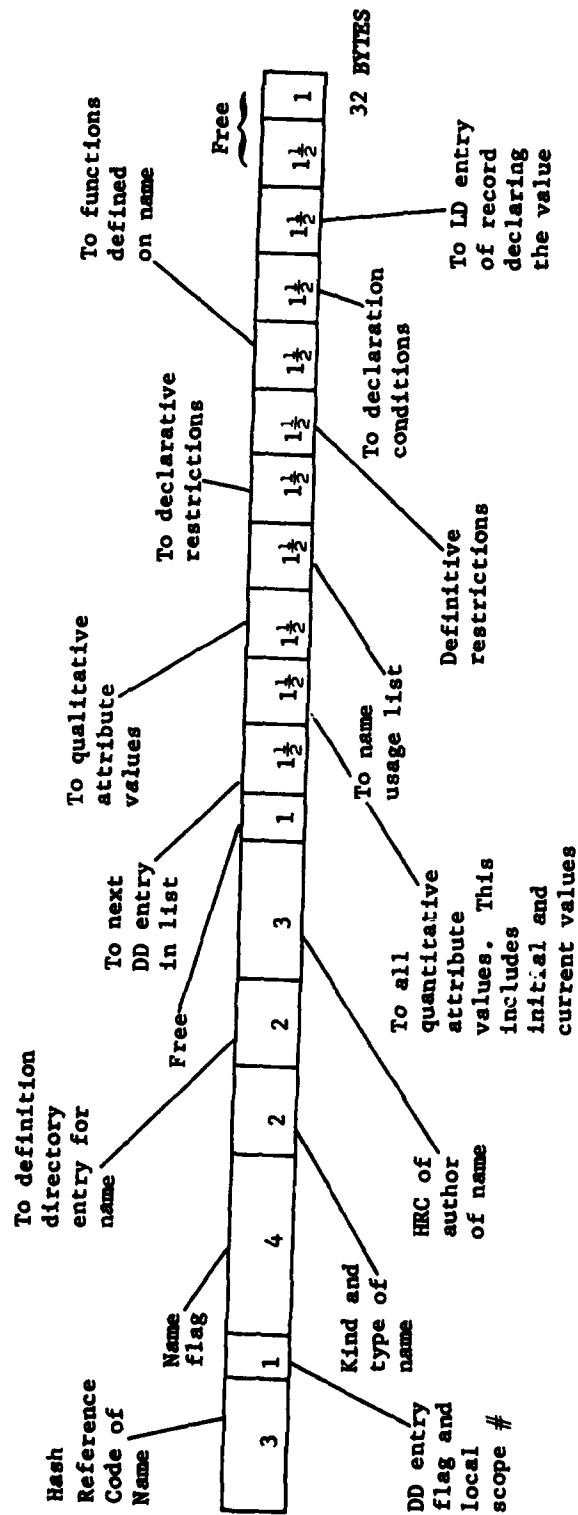
Figure 9(a). Declarations Directory Entry For a Declaration Without Symbolic Equality (A Name and Not an Alias)
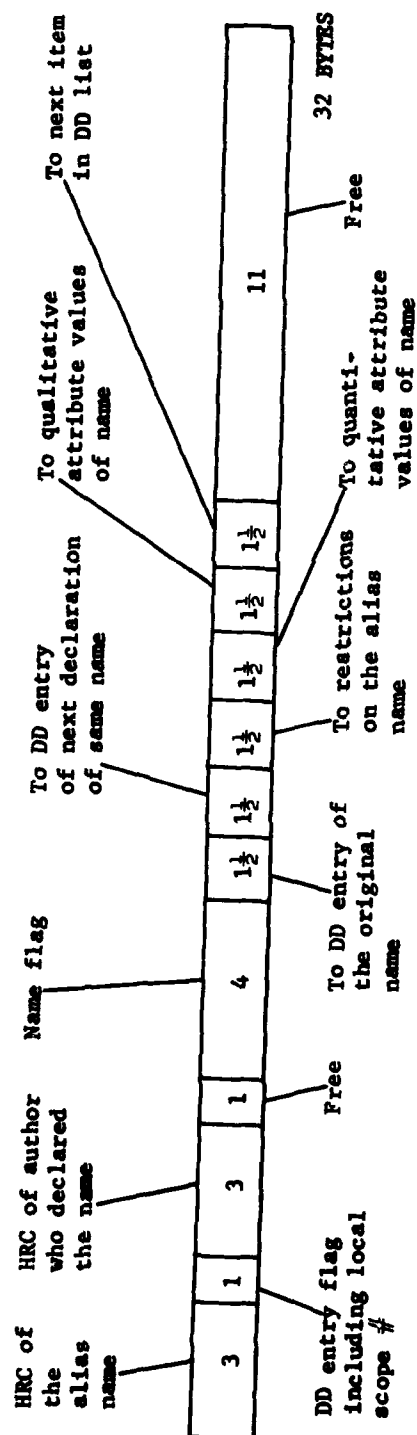
21

Figure 9(b). SDD Entry Format for a Name Which is an Alias for Some Other Name

REFERENCES

1. C. V. Srinivasan, CDL1, A Computer Description Language, "Part I: The Nature of the Description Language and Organization of Descriptions", and "Part II: Kinds of Descriptions of A Computing System", Scientific Report No. 3, AFCRL-69-0322, Contract No. F19628-68-C-0070, July 1969.

2. C. V. Srinivasan, An Introduction to CDL1, A Computer Description Language, Scientific Report No. 1, AFCRL-67-0565, Contract No. AF19(628)4789, September 1967.

3. C. V. Srinivasan, Formal Definition of CDL1, A Computer Description Language, Scientific Report No. 2, AFCRL-67-0588, Contract No. AF19(638)4789, October 1967.

4. A. J. Korenjak, "A Practical Method for Constructing LR(k) Processors", Com. ACM, November 1969.

5. A. J. Korenjak and D. A. Walters, Modeling Deterministic Syntax Analyzers by Reduction Procedures, Scientific Report No. 3, Contract No. F44620-68-C-0012, September 1968.

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
RCA Laboratories Princeton, New Jersey 08540	Unclassified
	2b. GROUP N/A

3. REPORT TITLE

ON THE IMPLEMENTATION OF THE DESCRIPTIVE DATA BASE, BASED ON CDL1

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
Scientific.   Interim.

5. AUTHOR(S) (First name, middle initial, last name)

Chitoor V. Srinivasan

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
February 1970	30	5

8a. CONTRACT OR GRANT NO. F19628-68-C-0070	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO. Project, Task, Work Unit Nos. 5632-02-01	Scientific Report No. 4
c. DoD Element 61102F	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d. DoD Subelement 681305	AFCRL-70-0184

10. DISTRIBUTION STATEMENT

This document has been approved for public release
and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
TECH, OTHER	Air Force Cambridge Research Laboratories L. G. Hanscom Field (CRB) Bedford, Massachusetts 01730

13. ABSTRACT

In previous reports, CDL1 -- a computer description language -- has been defined and discussed. This report discusses the implementation of a system of programs, on the RCA Spectra 70 computers, to generate appropriate file structures from computer descriptions written in CDL1. This translation to a DDB -- descriptive data base -- involves syntactic analysis and a certain amount of checking for internal consistency, as well as the creation of directory entries, etc. Once the type of DDB's described in this report can be generated, a variety of design-aid systems can be based upon them, saving a duplication of effort, guaranteeing an integrated overall system, and avoiding built-in obsolescence.

The final state of the work done under this contract is given, with details of the file and directory structures which have been implemented.

DD FORM 1473
1 NOV 68

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Design aid systems Computer description language Documentation Simulation Automatic synthesis						